Final Term Project: Option 1 - Supervised Data Mining (Classification)

Ryan Tolboom

Options

Classification algorithms used:

- Category 3 (Decision Trees) - C4.5
- Category 5 (Naive Bayes) - NaiveBayes

Dataset: Breast Cancer Wisconsin (Diagnostic) Data Set

Both algorithms were implemented by the author in Python for this project.

Preparation of Dataset

The dataset includes 683 complete instances. In preparation for the 10-fold cross validation method it was broken into ten parts and instances with unknown attribute values were removed. Concatenations of the nine remaining parts for each individual part were created to use as training data. The naming format used was `breast-cancer-wisconsin-part-xx` and `breast-cancer-wisconsin-part-xx.train`, where `xx` signifies the part number and the `.train` suffix is used to indicated which is the training data. The commands used to create these files in bash are shown below:

```
$ grep -v "?" breast-cancer-wisconsin.data | wc -l
683
$ grep -v "?" breast-cancer-wisconsin.data | \
> split -d -l 69 - breast-cancer-wisconsin-part-
$ for FILE in *part*; GLOBIGNORE="$FILE:*.train"; cat *part* > $FILE.train; done
$ ls -sh1 *part*
4.0K breast-cancer-wisconsin-part-00
 32K breast-cancer-wisconsin-part-00.train
4.0K breast-cancer-wisconsin-part-01
 32K breast-cancer-wisconsin-part-01.train
4.0K breast-cancer-wisconsin-part-02
 32K breast-cancer-wisconsin-part-02.train
4.0K breast-cancer-wisconsin-part-03
 32K breast-cancer-wisconsin-part-03.train
4.0K breast-cancer-wisconsin-part-04
```

```
 20K breast-cancer-wisconsin-part-04.train
4.0K breast-cancer-wisconsin-part-05
 20K breast-cancer-wisconsin-part-05.train
4.0K breast-cancer-wisconsin-part-06
 32K breast-cancer-wisconsin-part-06.train
4.0K breast-cancer-wisconsin-part-07
 20K breast-cancer-wisconsin-part-07.train
4.0K breast-cancer-wisconsin-part-08
 20K breast-cancer-wisconsin-part-08.train
4.0K breast-cancer-wisconsin-part-09
 20K breast-cancer-wisconsin-part-09.train
```

It should be noted that the data uses a `4` in the last column to indicate that the cells are malignant. This was used as the `True` class in our binary classification. All other entries are marked with `2` for benign, which corresponds to our `False` class. Lastly, the first column is a numeric identifier for each entry, which is unused.

C4.5 Decision Tree

An implementation of the C4.5 decision tree algorithm was developed in Python based on notes from Dr. Jason Wang and the writings of J. R. Quinlan. The first argument is a set of training data to use and the second argument is the test data to predict labels for. The output is a count of true positives, false positives, true negatives, and false negatives.

Source Code

```
########################################
###### Decision Tree in Python ########
########################################
# Usage is python3 decision_tree.py <training data> <test data>

import math
import pprint
import csv
import sys

pp = pprint.PrettyPrinter(indent=4)

def entropy(p, n):
    """
    Calculates the entropy for one attribute using the number of occurrences
    """

    # This avoids some log(0) issues
    if p == 0 or n == 0:
        return 0
```

```
    # Convert from a number of occurrences to a probability
    p1 = p / (p + n)
    n1 = n / (p + n)

    return -1 * p1 * math.log(p1, 2) - n1 * math.log(n1, 2)

def entropy2(counts):
    """
    Calculates the entropy for two attributes given a list of [p, n]
    """

    # Calculate the total
    total = 0
    for row in counts:
        total += row[0] + row[1]

    # Calculate the entropy for the two attributes
    entropy2 = 0
    for row in counts:
        p = row[0]
        n = row[1]
        occurrences = p + n
        entropy2 += occurrences / total * entropy(p, n)
    return entropy2

def select_attribute(data):
    """
    Chooses the attribute, by index, that would net the greatest information
    gain. Assumes the last entry in a row is the boolean class
    """

    # Calculate the total entropy
    p = 0
    n = 0
    for row in data:
        if row[-1]:
            p += 1
        else:
            n += 1
    total_entropy = entropy(p, n)

    # Calculate the gain for each attribute and keep track of the index of the
    # max
    max_index = 0
    max_gain = 0
    for index in range(len(data[0]) - 1):
        counts = {}
```

```
        for row in data:
            attribute_value = row[index]
            if attribute_value not in counts:
                counts[attribute_value] = [0, 0]
            if row[-1]:
                counts[attribute_value][0] += 1
            else:
                counts[attribute_value][1] += 1
        gain = total_entropy - entropy2(counts.values())
        #print(f"Index: {index} Gain: {gain}")
        if gain > max_gain:
            max_gain = gain
            max_index = index

    return max_index

def split_on_attribute(data, index):
    """
    Takes the current data and breaks it into groups based by an attribute index
    This attribute is removed from the resulting groups
    """
    groups = {}
    for row in data:
        attribute_value = row[index]
        # Since we are already making a decision based on this attribute we don't
        # need to include it in the lower nodes
        row.pop(index)
        if attribute_value not in groups:
            groups[attribute_value] = []
        groups[attribute_value].append(row)
    return groups

def majority_class(data):
    """
    Counts the classes of the rows in a dataset to determine if they are
    unanimous and what the majority is
    """
    p = 0
    n = 0
    for row in data:
        if row[-1]:
            p += 1
        else:
            n += 1
    result = {}
    result['unanimous'] = True if (p == 0 or n == 0) else False
    result['majority'] = p > n
    return result
```

```
class Node:
    """
    A general class for the nodes of a tree
    """
    type = None
    index = None
    majority = None
    branches = None

def build_tree(data):
    """
    Builds a decision tree from a dataset
    """
    #print("Creating node from data...")
    #pp.pprint(data)
    node = Node()

    # Check to see if all the labels are the same, if so we are creating a RESULT
    # node
    result = majority_class(data)
    node.majority = result['majority']
    if result['unanimous']:
        #print(f"RESULT: {result['majority']}")
        node.type = 'RESULT'
        return node

    # If not we are creating a DECISION node
    node.type = 'DECISION'
    index = select_attribute(data)
    node.index = index
    node.branches = {}
    #print(f"DECISION: Splitting on index {index}...")
    groups = split_on_attribute(data, index)
    for attribute_value, group_data in groups.items():
        #print(f"Creating {attribute_value} node")
        node.branches[attribute_value] = build_tree(group_data)
    return node

# Testing data for tree building
#
#data = [
#    ['<75',     '1', 'City', 'MCI',    True],
#    ['75..150', '2', 'Town', 'MCI',    True],
#    ['<75',     '1', 'City', 'Sprint', False],
#    ['>150',    '2', 'Town', 'AT&T',   True],
#    ['75..150', '1', 'City', 'MCI',    False],
#    ['75..150', '2', 'Town', 'AT&T',   True],
```

```
#    ['<75',     '2', 'Town', 'AT&T',   False],
#    ['>150',    '2', 'City', 'Sprint', True],
#    ['<75',     '1', 'City', 'AT&T',   False],
#    ['75..150', '1', 'Town', 'MCI',    True],
#    ['75..150', '2', 'City', 'Sprint', True],
#    ['>150',    '1', 'Town', 'AT&T',   True],
#    ['<75',     '1', 'Town', 'AT&T',   False],
#    ['<75',     '2', 'City', 'Sprint', False],
#    ['75..150', '2', 'City', 'MCI',    True],
#]
#root = build_tree(data)
#import pdb; pdb.set_trace()

def classify_row(row, node):
    """
    Uses a decision tree to classify a row
    """
    if node.type == 'RESULT':
        return node.majority
    else:
        attribute_value = row[node.index]
        if attribute_value not in node.branches:
            #print("Unable to find unanimous result, using majority")
            return node.majority
        else:
            return classify_row(row, node.branches[attribute_value])

# Load the training data from the file
data = []
with open(sys.argv[1], newline='') as csvfile:
    trainingdata = csv.reader(csvfile)
    for row in trainingdata:
        # Our training data uses 2 for benign and '4' for malignant
        if row[10] == '4':
            row[10] = True
        else:
            row[10] = False
        # We don't need the first column, it is an ID
        data.append(row[1:])

# Recursively create the decision tree from the data
root = build_tree(data)

# Load the test data and classify each row keep track of TP, TN, FP, FN
results = { 'TP': 0, 'TN': 0, 'FP': 0, 'FN': 0 }
with open(sys.argv[2], newline='') as csvfile:
    testdata = csv.reader(csvfile)
    for row in testdata:
```

```
        if classify_row(row[1:9], root): # Positive
            if row[10] == '4': # True positive
                results['TP'] += 1
            else: # False positive
                results['FP'] += 1
        else: # Negative
            if row[10] == '2': # True negative
                results['TN'] += 1
            else: # False negative
                results['FN'] += 1

print(results)
```

Output

The program was run in a loop on all ten parts created earlier. The command to run the program as well as the program's output can be see below:

```
$ for PART in 00 01 02 03 04 05 06 07 08 09
> do python3 ./decision_tree.py \
> breast-cancer-wisconsin/breast-cancer-wisconsin-part-$PART.train \
> breast-cancer-wisconsin/breast-cancer-wisconsin-part-$PART
> done
{'TP': 26, 'TN': 33, 'FP': 2, 'FN': 8}
{'TP': 22, 'TN': 42, 'FP': 0, 'FN': 5}
{'TP': 26, 'TN': 40, 'FP': 1, 'FN': 2}
{'TP': 34, 'TN': 27, 'FP': 3, 'FN': 5}
{'TP': 27, 'TN': 33, 'FP': 3, 'FN': 6}
{'TP': 15, 'TN': 50, 'FP': 4, 'FN': 0}
{'TP': 18, 'TN': 48, 'FP': 2, 'FN': 1}
{'TP': 10, 'TN': 59, 'FP': 0, 'FN': 0}
{'TP': 21, 'TN': 45, 'FP': 1, 'FN': 2}
{'TP': 11, 'TN': 49, 'FP': 2, 'FN': 0}
```

Average Results over 10-Fold Cross Validation

True Positive	True Negative	False Positive	False Negative
21.0	42.6	1.8	2.9

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{21.0 + 42.6}{21.0 + 42.6 + 1.8 + 2.9} = 0.93$$

$$Precision = \frac{TP}{TP + FP} = \frac{21.0}{21.0 + 1.8} = 0.92$$

$$Recall = \frac{TP}{TP + FN} = \frac{21.0}{21.0 + 2.9} = 0.88$$

Naive Bayes

An implementation of the naive bayes algorithm was developed in Python. The first argument is a set of training data to use and the second argument is the test data to predict labels for. The output is a count of true positives, false positives, true negatives, and false negatives.

Source Code

```
#######################################################
########## Naive Bayes in Python ###################
#######################################################
# Usage is: python3 naive_bayes.py <training_data> <test_data>

import sys
import math
import csv
import pprint
pp = pprint.PrettyPrinter(indent=4)

# Training

# Initialize the data
means = {
    'benign'   : [0,0,0,0,0,0,0,0,0],
    'malignant': [0,0,0,0,0,0,0,0,0],
}
counts = {
    'benign'   : 0,
    'malignant': 0,
}
stddevs = {
    'benign'   : [0,0,0,0,0,0,0,0,0],
    'malignant': [0,0,0,0,0,0,0,0,0],
}

# Read from the training file
with open(sys.argv[1], newline='') as csvfile:
    trainingdata = csv.reader(csvfile)

    # Sum up the columns for each class and count instances
    for row in trainingdata:
        # There are two classes in this dataset 2: benign and 4: malignant
        if row[10] == '2':
            label = 'benign'
        else:
            label = 'malignant'
        counts[label] += 1
```

```
            # The first column is an ID which we don't need for training
            for col in range(1, 10):
                means[label][col - 1] += int(row[col])

    # Divide by the count to get the mean
    for label in ['benign', 'malignant']:
        for col in range(9):
            means[label][col] = means[label][col] / counts[label]

    # Calculate the standard deviation
    # Iterate through the dataset again
    csvfile.seek(0)
    for row in trainingdata:
        # There are two classes in this dataset 2: benign and 4: malignant
        if row[10] == '2':
            label = 'benign'
        else:
            label = 'malignant'
        # The first column is an ID which we don't need for training
        # sum the squares of the distance from the mean
        for col in range(1, 10):
            stddevs[label][col - 1] += (int(row[col]) - means[label][col - 1])**2
    # divide by the count to get the average and take the square root
    for label in ['benign', 'malignant']:
        for col in range(9):
            stddevs[label][col] = math.sqrt(stddevs[label][col] / counts[label])

# Prediction

results = {'TP': 0, 'TN': 0, 'FP': 0, 'FN': 0}

with open(sys.argv[2], newline='') as csvfile:
    testdata = csv.reader(csvfile)
    for row in testdata:
        distances = {
            'benign':    0,
            'malignant': 0,
        }
        for col in range(1, 10):
            for label in ['benign', 'malignant']:
                # Calculate the squared distance normalized by stddev for each
                distances[label] += ((means[label][col - 1] - int(row[col])) /
                                     stddevs[label][col - 1])**2
        if distances['malignant'] < distances['benign']:
            # A positive prediction
            if row[10] == '4':
                # True positive
                results['TP'] += 1
```

```
            else:
                # False positive
                results['FP'] += 1
        else:
            # A negative prediction
            if row[10] == '2':
                # True negative
                results['TN'] += 1
            else:
                # False negative
                results['FN'] += 1

print(results)
```

Output

The program was run in a loop on all ten parts created earlier. The command to run the program as well as the program's output can be see below:

```
$ for PART in 00 01 02 03 04 05 06 07 08 09
> do
> python3 ./naive_bayes.py \
> breast-cancer-wisconsin/breast-cancer-wisconsin-part-$PART.train \
> breast-cancer-wisconsin/breast-cancer-wisconsin-part-$PART
> done
{'TP': 34, 'TN': 30, 'FP': 5, 'FN': 0}
{'TP': 27, 'TN': 36, 'FP': 6, 'FN': 0}
{'TP': 28, 'TN': 38, 'FP': 3, 'FN': 0}
{'TP': 38, 'TN': 25, 'FP': 5, 'FN': 1}
{'TP': 33, 'TN': 32, 'FP': 4, 'FN': 0}
{'TP': 15, 'TN': 49, 'FP': 5, 'FN': 0}
{'TP': 19, 'TN': 45, 'FP': 5, 'FN': 0}
{'TP': 10, 'TN': 57, 'FP': 2, 'FN': 0}
{'TP': 23, 'TN': 44, 'FP': 2, 'FN': 0}
{'TP': 11, 'TN': 49, 'FP': 2, 'FN': 0}
```

Average Results over 10-Fold Cross Validation

True Positive	True Negative	False Positive	False Negative
23.8	40.5	3.9	0.1

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{23.8 + 40.5}{23.8 + 40.5 + 3.9 + 0.1} = 0.94$$

$$Precision = \frac{TP}{TP + FP} = \frac{23.8}{23.8 + 3.9} = 0.86$$

$$Recall = \frac{TP}{TP + FN} = \frac{23.8}{23.8 + 0.1} = 1.00$$

Analysis

Algorithm	Accuracy	Precision	Recall
Decision Tree	0.93	0.92	0.88
Naive Bayes	0.94	0.86	1.00

As can be seen, in terms of accuracy the naive bayes classifier slightly outperformed the decision tree classifier. That being said accuracy may not be the most useful statistic to use. This particular dataset is not symmetric, 65.5% of the samples are for benign cells and 34.4% of the samples are for malignant cells. Accuracy weights both true positives and true negatives equally and as such may be misleading.

The decision tree classifier was significantly more precise than the naive bayes classifier meaning that when the decision tree classifier predicted that cells were malignant it was more likely that it was correct. This due to its low false positive rate of 1.8 compared to the naive bayes rate of 3.9.

The naive bayes classifier has a much higher recall score than the decision tree classifier. The naive bayes classifier actually achieved a perfect score, within the rounding margins, for this statistic meaning that it did not miss any of the cells that should have been labeled malignant. The false negative rate of the naive bayes classifier was a mere 0.1.

Given the nature of the data, the naive bayes classifier would be the best model to use. A high recall rate is critical when identifying cancerous tissue as the penalty for false negatives is significantly higher than the penalty for false positives. A false negative may prevent a patient from seeking early treatment, while a false positive just leads to further examinations.

Ideas for Improvement

The C4.5 algorithm actually has the ability to handle training data with missing attributes by leaving them out of gain and entropy calculations. This was not used in this analysis as the naive bayes algorithm was not able to handle such training data. One major advantage of implementing this would be an increase in the amount of training data available to the decision tree. Given the nature of its construction, more data available during the creation of the tree results in more paths being created and less of a chance of the test data ending at a decision for which there is no known branch.

It should also be noted that the greedy nature of the C4.5 algorithm does not lend itself well to the map-reduce techniques often used for dealing with large datasets. While it is true that the algorithm continually segments the data into smaller and smaller chunks, each step requires going through all of the data passed to it to calculate the entropy, and the tree is built from the top down. The naive bayes classifier better lends itself to large data sets as its operations, namely the calculation of means and standard deviations, can be done on chunks of the data and then reduced together.

Sources

- Quinlan, J. R. 1986. Induction of Decision Trees. Mach. Learn. 1, 1 (Mar. 1986), 81–106
- Quinlan, J. R. C4.5: Programs for Machine Learning. Morgan Kaufmann Publishers, 1993.
- Wang, Jason T. L. Data Mining Classification I - Decision Trees (Part B) [Powerpoint Slides]. Retrieved from https://lmscontent.embanet.com/NJIT/CS634/documents/CS634-w03-m01B.pdf
- Wolberg, William H. University of Wisconsin Hospitals: Breast Cancer Database. Retrieved from http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29
- O. L. Mangasarian and W. H. Wolberg: "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, September 1990, pp 1 & 18.

All source code listed, as well as the markdown used to create this paper are available from https://github.com/rxt1077/CS634